CS383 Project - Part 1

Cecilia Chen
Bryn Mawr College
USA
zchen4@brynmawr.edu

Cecilia Zhang
Bryn Mawr College
USA
czhang6@brynmawr.edu

Tianyun Song
Bryn Mawr College
USA
tsong@brynmawr.edu

Yang Wu
Bryn Mawr College
USA
ywu3@brynmawr.edu

ABSTRACT

This paper evaluates the effectiveness of EvoSuite and EditAS2 in detecting bugs in the Defects4J Time project (bugs 1–12). While EvoSuite achieved 100% prefix coverage and an average branch coverage of 94.2%, EditAS2 generated assertions for only 6 of the 12 bugs, with just 2 resulting in successful bug detection. Our analysis reveals that failures were often due to a lack of domain-relevant inputs (e.g., leap-year dates) or missing runtime check assertions (e.g., `assertThrows`). Bugs requiring semantic reasoning or multithreaded contexts also remained undetected. To address these limitations, we propose domain-aware input seeding, fallback exception templates, and leveraging developer-written tests to guide assertion and prefix generation.

ACM Reference Format:
Cecilia Chen, Cecilia Zhang, Tianyun Song, and Yang Wu. 2025. CS383 Project - Part 1. In . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This study investigates the capabilities of state-of-the-art automated test generation tools in detecting real-world software defects. We focus our analysis on the bugs 1-12 in Defects4J Time project. The Time project provides an ideal testbed as it contains a mix of safety property violations (e.g., improper exception handling) and functional property violations (e.g., incorrect time calculations). Our evaluation specifically assesses EvoSuite's ability to generate test prefixes that execute buggy code paths and EditAS2's effectiveness in producing meaningful assertions that expose faults. Through this systematic examination, we aim to identify the strengths and limitations of current automated testing approaches when applied to real-world temporal logic implementations.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Bug Number	Safety or Functional Property
1	Safety
2	Safety
3	Functional
4	Safety
5	Functional
6	Safety/Functional
7	Safety
8	Safety
9	Functional
10	Safety
11	Functional
12	Functional

Table 1: Developer Written Tests.

```
public Partial(DateTimeFieldType[] types, int[] values,
Chronology chronology) {
   ...
   int compare = lastUnitField.compareTo(loopUnitField);
   if (compare < 0) {
       // exception
   } else if ( compare == 0)
   {
           ...
       }
   }
}

public int compareTo(DurationField durationField) {
   if (durationField.isSupported()) {
       return 1;
   }
       return 0;
}
```

Figure 1: Defects4J Time1 Bug. Constructor throws IllegalArgumentException when passed duplicate DateTimeFieldTypes.

Time-1 concerns a defect in the `Partial` constructor of the Joda-Time library, where input validation for `DateTimeFieldType` arrays is incorrectly implemented. The constructor is expected to throw an `IllegalArgumentException` if the field types are not in decreasing order of duration or if duplicates exist. In the buggy version, these checks are missing, allowing invalid inputs to proceed

silently. Additionally, the constructor attempts to sort field types using their RangeDurationFieldType, but fails to handle null values (e.g., for fields like year), resulting in a NullPointerException.

```
public Partial(DateTimeFieldType[] types, int[] values,
Chronology chronology) {
   DurationField loopUnitField =
   loopType.getDurationType().getField(iChronology);
   if (i > 0) {
       int compare = lastUnitField.compareTo(loopUnitField);
       if (compare < 0
       || (compare != 0 && loopUnitField.isSupported() == false)) {
           throw new IllegalArgumentException(
           "Types array must be in order largest-smallest: "
           + types[i - 1].getName() + " < " + loopType.getName());
       }
   ...
}

public Partial with(DateTimeFieldType fieldType, int value) {
   ...
                   if (compare > 0) {
                       break;
                   } else if (compare == 0) {
                       DurationField rangeField = fieldType
                       .getRangeDurationType().getField(iChronology);
                       DurationField loopRangeField = loopType.
                       getRangeDurationType().getField(iChronology);

   ...
}
```

Figure 2: Defects4J Time2 Bug. The `with()` method throws when inserting a logically overlapping field.

Time-2 concerns the `with()` method, which updates a `Partial` with a new field. When the new field overlaps logically with an existing one (e.g., `hourOfHalfday` and `clockhourOfHalfday`), an `IllegalArgumentException` should be thrown. However, in the buggy version, certain conflicting combinations were allowed.

This failure occurs in the constructor that `with()` internally calls to build the updated `Partial`. The constructor's validation logic is responsible for detecting invalid field combinations, as shown in Figure 17.

```
public void addDays(final int days) {
   // Bug: Missing no-op check for zero input
   setMillis(getChronology().days().add(getMillis(), days));
}

// Patch version adds:
public void addDays(final int days) {
   if (days != 0) {
       setMillis(getChronology().days().add(getMillis(), days));
   }
}
```

Figure 3: Defects4J Time3 Bug. `addDays(0)` should be a no-op but incorrectly shifts time during DST.

Time-3 concerns a defect in the `MutableDateTime` class, where several `addX()` methods (e.g., `addDays`, `addMonths`, `addYears`) failed to handle zero-input values correctly. When called with an argument of zero during a Daylight Saving Time (DST) overlap, these methods unnecessarily modified the internal time state due to improper time zone normalization, resulting in incorrect shifts (e.g., from `+01:00` to `+02:00`). The fix added explicit zero-checks to prevent such unintended changes.

```
public Partial with(DateTimeFieldType fieldType, int value) {
       ...
       // use public constructor to ensure full validation
       // this isn't overly efficient, but is safe
       Partial newPartial = new Partial(
           iChronology, newTypes, newValues);
       iChronology.validate(newPartial, newValues);
       return newPartial;
   }
       ...
}
```

Figure 4: Defects4J bug Time4. Uses the contructor that does not have validation to prevent DateTimeFieldTypes that are semantically overlapping.

Time-4 occurs in the `Partial.with()` method. Calling the with method can set a field of a `Partial` to a specified value. Inside `with()`, when adding a new `DateTimeFieldType` to the `Partial`, the method calls a constructor of `Partial(Chronology chronology, DateTimeFieldType[] types, int[] values)`, which fails to properly validate and reject cases where the new field has the same unit with an existing one. For example, adding `minuteOfHour` to a `Partial` that already has the field `minuteOfDay` is not semantically valid. However, since this constructor does not have any validation, such duplicates are allowed, leading to potential data integrity issues with the new `Partial` object.

```
public Period normalizedStandard(PeriodType type) {
   ...
   int years = getYears();
   int months = getMonths();
   if (years != 0 || months != 0) {
       years = FieldUtils.safeAdd(years, months / 12);
       months = months % 12;
       if (years != 0) {
           result = result.withYears(years);
       }
       if (months != 0) {
           result = result.withMonths(months);
       }
   }
   return result;
}
```

Figure 5: Defects4J bug Time5. Attempts to access fields that are not supported.

Time-5 occurs in the `normalizedStandard(PeriodType type)` method of the `Period` class. It attempts to normalize `year` and `month` fields without properly checking if the `PeriodType` supports these fields. This could lead to runtime exceptions when the `Period` include `month` or `year`, but the type did not support one of them.

```
public long add(long instant, long value) {
   ...
       if (instant < iCutover) {
           // Only adjust if gap fully crossed.
           if (instant + iGapDuration < iCutover) {
               instant = gregorianToJulian(instant);
   ...
}
```

Figure 6: Defects4J bug Time6. `add(...)` handles cutover date incorrectly.

Time-6 is in the `GJChronology.add(...)` method, which handles date arithmetic across the Julian-Gregorian cutover. The bug occurs when adding a value to an instant that crosses the cutover point into year 0 or earlier, which is not valid in the Gregorian calendar. Without additional handling, this could lead to the generation of a wrong or invalid Gregorian date.

```
public int parseInto(ReadWritableInstant instant, String text,
        int position) {
        ...

        long instantMillis = instant.getMillis();
        Chronology chrono = instant.getChronology();
        long instantLocal = instantMillis +
                chrono.getZone().getOffset(instantMillis);
        chrono = selectChronology(chrono);

        int defaultYear = chrono.year().get(instantLocal);
        ...
    }
```

Figure 7: Defects4J Time7 Bug. Feb 29th for non-leap years using certain time zones throws an exception.

Time-7 occurs when attempting to parse February 29 in a non-leap year using certain time zones. The parser fails to validate the date correctly and throws an `IllegalFieldValueException`. This is a **safety property violation**.

```
public static DateTimeZone forOffsetHoursMinutes(int
hoursOffset, int minutesOffset) throws IllegalArgumentException {
    ...
    int hoursInMinutes = hoursOffset * 60;
    if (hoursInMinutes < 0) {
        minutesOffset = hoursInMinutes - Math.abs(minutesOffset);
    } else {
        minutesOffset = hoursInMinutes + minutesOffset;
    }
    offset = FieldUtils.safeMultiply(minutesOffset,
        DateTimeConstants.MILLIS_PER_MINUTE);
    return forOffsetMillis(offset);
}
```

Figure 8: Defects4J Time8 Bug. A safety bug triggered by invalid minute input.

Time-8 is a safety property violation in the method `forOffsetHoursMinutes(int hours, int minutes)`. This method is expected to throw an `IllegalArgumentException` when the `minutes` argument is outside the valid range of `-59` to `+59`. In the buggy version, this input validation was missing, allowing extremely large or small values to pass unchecked. For example, passing a value like `-75837680` for minutes led to incorrect internal state or crashes.

```
public static DateTimeZone forOffsetHoursMinutes(int
hoursOffset, int minutesOffset) {
    ...
    int offset = 0;
    try {
        int hoursInMinutes = hoursOffset * 60;
        if (hoursInMinutes < 0) {
            minutesOffset = hoursInMinutes - Math.abs(minutesOffset);
        } else {
            minutesOffset = hoursInMinutes + minutesOffset;
        }
        offset = FieldUtils.safeMultiply(minutesOffset,
        DateTimeConstants.MILLIS_PER_MINUTE);
    } catch (ArithmeticException ex) {
        throw new IllegalArgumentException("Offset is too large");
    }
    return forOffsetMillis(offset);
}
```

Figure 9: Defects4J Time9 Bug. The `forOffsetHoursMinutes()` accepts invalid inputs like `(45, 45)` and computes a malformed offset instead of throwing an exception.

Time-9 is a **functional property violation** in the method `forOffsetHoursMinutes(int hours, int minutes)`. Instead of throwing an exception for invalid input combinations, the buggy implementation computes and returns a `DateTimeZone` object with an incorrect internal offset. For example, calling the method with inputs like `(25, 61)` or other out-of-range values does not result in a failure, but instead returns a malformed zone such as `"UTC+26:01"`. Unlike Time-8, this bug does not manifest as a crash, making it detectable only through output assertions rather than exceptions.

```
protected static int between(ReadablePartial start,
    ReadablePartial end, ReadablePeriod zeroInstance) {

    // throw illegal argument
    Chronology chrono = DateTimeUtils.
        getChronology(start.getChronology()).withUTC();
    int[] values = chrono.get(zeroInstance,
                        chrono.set(start, 0L),
                            chrono.set(end, 0L));
    return values[0];
    }
```

Figure 10: Defects4J Time10 Bug. A functional bug that manifests as an IllegalArgumentException.

Time-10 initializes the chrono's start to be $0L$ without starting from 1972, which is the leap year starting on Saturday of the Gregorian calendar. This will cause the IllegalFieldValueException if we have Day 29 for February, making this a **safety property violation**. The bug will be triggered when we call `monthsBetween()` and `daysBetween()` when the first parameter's type is `ReadablePartial` with input date as 02/29.

```
public class ZoneInfoCompiler {
    ...
    static DateTimeOfYear cStartOfYear;

    static Chronology cLenientISO;

    static ThreadLocal<Boolean> cVerbose = new ThreadLocal<Boolean>();

    static {
        cVerbose.set(Boolean.FALSE);
    }
    ...
}
```

Figure 11: Defects4J Time11 Bug.

Time-11 initializes `cVerbose` as a `ThreadLocal<Boolean>` and sets it to `Boolean.FALSE` in a static block as shown in Figure 11. Since this only affects the current thread, other threads receive `null`, causing failures like `AssertionFailedError` in `TestCompiler:testDateTimeZoneBuilder`. The fix overrides `initialValue()` to return `Boolean.FALSE` to handle the case of double thread, making this a **functional property violation**. This will be triggered by any input that calls the method `.toDateTimeZone()` inside the double thread.

```
public static LocalDate fromCalendarFields(Calendar calendar) {
    ...
    int yearOfEra = calendar.get(Calendar.YEAR);
    return new LocalDate(
        yearOfEra,
        calendar.get(Calendar.MONTH) + 1,
        calendar.get(Calendar.DAY_OF_MONTH)
        );
    ...
}
public static LocalDateTime fromCalendarFields(Calendar calendar) {
    ...
    int yearOfEra = calendar.get(Calendar.YEAR);
    return new LocalDateTime(
        yearOfEra,
        calendar.get(Calendar.MONTH) + 1,
        calendar.get(Calendar.DAY_OF_MONTH),
    ...
}
```

Figure 12: Defects4J Time12 Bug.

Time-12 mishandles BC dates by using the year without adjusting for era, causing incorrect results like `0001` instead of `0000` or `-0002`. The fix computes the proleptic year with `(AD ? year : 1 - year)` and uses `fromCalendarFields` for negative timestamps. This is a **functional property violation**. This fault will be triggered by calling method `.fromDateFields()` and `.fromCalendarFields()` for `LocalDate` and `LocalDateTime` such as `testFactory_fromDateFields_beforeYearZero1`.

```
public Partial(DateTimeFieldType[] types, int[] values,
    Chronology chronology) {
    this.iChronology = chronology;

    for (int i = 0; i < types.length; i++) {
        DateTimeFieldType loopType = types[i];
        DurationField loopUnitField = loopType.
        getDurationType().
        getField(iChronology);

        if (i > 0) {
            DurationField lastUnitField = types[i - 1].
            getDurationType().getField(iChronology);

            if (!loopUnitField.isSupported()) {
                if (lastUnitField.isSupported()) {
                    throw new IllegalArgumentException(
                    "Types array must be in order
                    largest-smallest: " +
                    types[i - 1].getName() +
                    " < " +
                    loopType.getName());
                } else {
                    throw new IllegalArgumentException(
                    "Types array must not contain
                    duplicate unsupported: " +
                    types[i - 1].getName() +
                    " and " + loopType.getName());
                }
            }

            int compare = lastUnitField.compareTo(loopUnitField);
            if (compare < 0) {
                throw new IllegalArgumentException(
                "Types array must be in order
                largest-smallest: " + types[i - 1].
                getName() +
                " < " +
                loopType.getName());
            } else if (compare == 0 && lastUnitField.
```

Figure 13: Patched constructor for `Partial`. This version adds checks to prevent null, duplicate, or misordered `DateTimeFieldType` entries and throws an `IllegalArgumentException` accordingly.

2 EVALUATION

2.1 Prefix Generation

In our experiments with EvoSuite, we evaluated its ability to generate prefixes that execute the buggy methods for Time 1-12.

The bug in **Time-1** lies in the `Partial` constructor, which expects an array of `DateTimeFieldType` instances to follow a strict largest-to-smallest order, with no duplicates. Violating this contract—such as passing `dayOfMonth()`, `year()`, and `monthOfYear()` out of order—should result in an `IllegalArgumentException`.

This logic was incomplete in the buggy version. Certain invalid combinations (e.g., unsupported duplicate types) were not caught, resulting in incorrect behavior—**the constructor silently accepted invalid input**. The human patch (see Figure 13) adds these missing checks. The developer-written test `testConstructorEx7_TypeArray_intArray()` explicitly verifies that the constructor throws on such inputs (Figure 14).

EvoSuite successfully generated valid prefixes across all three seeds (Seed1–Seed3). Each generated test (e.g., `test18`, `test19`, `test20`) instantiated `Partial` with problematic field arrays, which—on the buggy version—executed without error. The generated assertions captured this faulty acceptance behavior.

Bug Number	Prefix Covered Bug	Assertion Triggered Bug	Bug Found
1	100.00%	0.00%	0.00%
2	100.00%	0.00%	0.00%
3	0.00%	0.00%	0.00%
4	100.00%	0.00%	0.00%
5	0.00%	0.00%	0.00%
6	100.00%	0.00%	0.00%
7	0.00%	0.00%	0.00%
8	0.00%	0.00%	0.00%
9	33.33%	0.00%	0.00%
10	100.00%	0.00%	0.00%
11	100.00%	0.00%	0.00%
12	100.00%	0.00%	0.00%

Table 2: Prefix and assertion result for each bug.

```
public void testConstructorEx7_TypeArray_intArray() throws Throwable {
   int[] values = new int[] {1, 1, 1};
   DateTimeFieldType[] types = new DateTimeFieldType[] {
       DateTimeFieldType.dayOfMonth(),
       DateTimeFieldType.year(),
       DateTimeFieldType.monthOfYear() };
   try {
       new Partial(types, values);
       fail();
   } catch (IllegalArgumentException ex) {
       assertMessageContains
       (ex, "must be in order",
       "largest-smallest");
   }

   \\ another try-catch testing
}
```

Figure 14: Developer-written test for Time-1: `testConstructorEx7_TypeArray_intArray()`. This test ensures invalid orderings throw an `IllegalArgumentException`.

```
public void test18() throws Throwable  {
   DateTimeFieldType dateTimeFieldType0 =
       DateTimeFieldType.secondOfMinute();
   DateTimeFieldType dateTimeFieldType1 =
       DateTimeFieldType.clockhourOfDay();
   DateTimeFieldType[] dateTimeFieldTypeArray0 =
       new DateTimeFieldType[4];
   dateTimeFieldTypeArray0[0] = dateTimeFieldType1;
   dateTimeFieldTypeArray0[1] = dateTimeFieldType0;
   dateTimeFieldTypeArray0[2] = dateTimeFieldType0;
   dateTimeFieldTypeArray0[3] = dateTimeFieldType1;
   int[] intArray0 = new int[4];
   Partial partial0 = null;
   try {
       partial0 = new Partial(dateTimeFieldTypeArray0, intArray0);
       fail("Expecting exception: IllegalArgumentException");
   } catch(IllegalArgumentException e) {
       verifyException("org.joda.time.Partial", e);
   }
}
```

Figure 15: EvoSuite-generated test for Time-1 (e.g., `test18`). The test calls the `Partial` constructor with unordered or duplicate fields, reliably triggering the bug.

To validate this, we ran the same tests on the fixed version. The assertions failed due to the expected `IllegalArgumentException`, confirming that the EvoSuite-generated tests reliably expose the bug (see Figure 15).

```
public void testWith_baseAndArgHaveNoRange() {
   Partial test = new Partial(DateTimeFieldType.year(), 1);
   Partial result = test.with(DateTimeFieldType.era(), 1);
   assertEquals(2, result.size());
   assertEquals(0, result.indexOf(DateTimeFieldType.era()));
   assertEquals(1, result.indexOf(DateTimeFieldType.year()));
}
```

Figure 16: Developer-written test for Time-2: `testWith_baseAndArgHaveNoRange()`. This test confirms that adding conflicting fields to a `Partial` should throw an exception.

```
public void test24()  throws Throwable  {
   DateTimeFieldType dateTimeFieldType0 =
       DateTimeFieldType.clockhourOfHalfday();
   Partial partial0 =
       new Partial(dateTimeFieldType0, 1, (Chronology) null);
   DateTimeFieldType dateTimeFieldType1 =
       DateTimeFieldType.hourOfHalfday();
   try {
       partial0.with(dateTimeFieldType1, 1);
       fail("Expecting exception: IllegalArgumentException");
   } catch(IllegalArgumentException e) {
       verifyException("org.joda.time.Partial", e);
   }
}
```

Figure 17: EvoSuite-generated test for Time-2. This test uses the `with()` method to insert a logically overlapping field into a `Partial`, successfully exposing the bug.

The bug in **Time-2** involves the `Partial.with()` method, which allows inserting additional fields into a `Partial` object. When the inserted field conflicts with an existing one (e.g., inserting both `clockhourOfHalfday()` and `hourOfHalfday()`), it should throw an `IllegalArgumentException`. However, in the buggy version, certain conflicting combinations were not properly validated.

The developer-written test `testWith_baseAndArgHaveNoRange()` demonstrates one such failure. It inserts a second field that overlaps with an existing one, which should raise an error (Figure 16).

EvoSuite successfully generated prefix tests in all three seeds. For example, `test24` from Seed1, `test20` from Seed2, and `test23`/`test45` from Seed3 invoked the `with()` method using overlapping field types. These tests exercised the buggy logic and reliably triggered exceptions. These exceptions are ultimately thrown by the `Partial` constructor, which is internally called by `with()` to construct a new instance with the updated fields.

Figure 17 shows an EvoSuite-generated prefix that inserts conflicting fields using the `with()` method.

The bug in **Time-3** is located in the `MutableDateTime` class. Methods such as `addYears(0)` and `addDays(0)` are intended to be no-ops when passed a zero value. However, in the buggy version, these methods mistakenly changed the internal timestamp—but only under specific conditions. In particular, this bug manifests when the `MutableDateTime` object represents a time during a daylight saving time (DST) overlap. In such cases, zero-value operations like `addYears(0)` unexpectedly alter the internal time, violating the expectation that zero should have no effect.

In the patched version, a guard condition was added to skip the internal update if the argument is zero (see Figure 18). A developer-written test confirms that the bug is triggered specifically when a method like `addHours(1)` transitions the time into a DST-overlap window, and is then followed by a zero-value call such as `addYears(0)`.

Although EvoSuite successfully executed the buggy methods across all three trials, it did not generate any test that triggered the fault. None of the generated tests constructed a `MutableDateTime` instance that fell within a DST-overlap window, nor did they produce the specific method sequence needed to reveal the timestamp shift. This indicates that while EvoSuite achieved structural coverage of the buggy code, it failed to synthesize the precise domain-aware conditions required to expose the bug.

```
public void testAddYears_int_dstOverlapWinter_addZero() {
    MutableDateTime dt = new MutableDateTime(
    "2007-10-30T02:30:00.000+01:00");
    dt.addHours(1); // Step into DST
    dt.addYears(0); // Should be no-op
    assertEquals(
    "2007-10-30T02:30:00.000+01:00",
    dt.toString());
}
```

Figure 18: Developer-written test for Time-3: `testAddYears_int_dstOverlapWinter_addZero()`. This test shows that calling `addHours(1)` followed by `addYears(0)` leads to an incorrect internal timestamp.

Coverage analysis using Defects4J confirms that the generated tests did execute the relevant methods. Specifically, all three EvoSuite seeds (Seed1–Seed3) invoked methods such as `addYears(0)`, `addDays(0)`, and `addMonths(0)`. However, none of these tests led to observable assertion failures related to DST shifts.

We also verified this result using the Defects4J test failure reports. While one generated test (`test77`) failed, it was due to an unrelated `UnsupportedOperationException` caused by the unsupported `eras` field—not a DST-related timestamp error.

This illustrates a key limitation in EvoSuite's prefix generation: despite reaching the buggy branch, the absence of DST-aware input synthesis prevented the tool from triggering the observable fault. As a result, prefix generation for Time-3 was only *partially* successful—it covered the relevant method, but did not trigger the bug.

```
public void test34()  throws Throwable  {
    ...
    LocalTime localTime0 = LocalTime.now((Chronology) zonedChronology0);
    Partial partial0 = new Partial(localTime0);
    DateTimeFieldType dateTimeFieldType0 = DateTimeFieldType.
                                           minuteOfDay();
    Partial partial1 = partial0.with(dateTimeFieldType0, 23);
    Assert.assertEquals(0, partial1.getValue()); // EditAS2 generated
}
```

Figure 19: Test that triggers bug Time4.

For **Time-4**, EvoSuite generated 54 tests in each trial with random seeds. In each trial, it successfully generated prefixes that executed the buggy branch that call the `Partial` constructor without validation. The tests cover various scenarios, including null handling, unsupported fields, and invalid indices. Since the assertions generated by Evosuite do not work, we inspected the code to see if any test have the input that can trigger the bug. It turned out that in all three trials, there are tests that have bug triggering input. That is, when calling the `with(...)` method, if the specified DateTimeFieldType has the same range with the range of an existing field of `Partial`. The `Partial` constructor without validation is then called, resulting in a `newPatial` with invalid fields.

In Figure19, the specified is `minuteOfDay`. The partial is created with `LocalTime`, which contains the field `minuteOfHour`. Since these two fields are semantically overlapping, this test can trigger the bug.

For **Time-5**, EvoSuite generated about 71-74 tests that test various `Period` methods, including addition, subtraction, and comparison of instances. In all three trials, the buggy branch was reached and executed. In order to trigger the bug, the input parameter of `normalizedStandard(...)` should be a `PeriodType` that doesn't support the field `year` or `month`. Then when the method is trying to access its `year` or `month` field, the bug is triggered. However, after inspecting all the tests that call `normalizedStandard(...)`, we found that none of the call have a bug triggering input. All the input are `PeriodTypes` that support `year` and `month`, so the bug is not triggered. By using the debug mode to generate the tests, we found that Evosuite never got close to a bug triggering input. It was always using the most basic inputs to call the method, so the bug is not detected even when the buggy method is invoked.

```java
public void test08()  throws Throwable  {
 BuddhistChronology buddhistChronology0 = BuddhistChronology
                                          .getInstanceUTC();
 Years years0 = Years.years(2057);
 long long0 = buddhistChronology0.add((ReadablePeriod) years0,
                                      (-62135596800014L), 2057);
 assertEquals(long0, years0.get().getTime()); // EditAS2 generated
}
```

Figure 20: Test that triggers bug Time6.

For **Time-6**, EvoSuite generated about 33-39 tests, interacting with the `GJChronology` class, including different time zones, cutover times, and period-based calculations. In all three trials, the buggy branch is executed. In two of the three trials, there is bug triggering input. That is, when the calculation the `add(...)` method is doing, computes a millisecond timestamp spanning from a time before year 0 to a time after 1582.

Figure 20 shows a test that triggers the bug Time6. It is using BuddhistChronology that inherits the system from `GJChronology`, so it uses the same cutover date. The test adds the timestamp from -62135596800014L (which is a little earlier than year 0) to 133463186179199986L (which is way after year 1582). The `add(...)` called here is not the buggy method, but inside it the buggy `add(long instant, long value)` is called, so the bug is triggered.

```java
public void test14()  throws Throwable  {
    ...
    MutableDateTime mutableDateTime0 = dateTimeFormatter1.
      parseMutableDateTime("/2i({p{]:$bw");
    int int0 = dateTimeFormatter1.parseInto(mutableDateTime0,
      "/2i({p{]:$bw", 4);
    ...
}
```

Figure 21: Test generated by Evo-suite for Time-7. Input does not match any date format.

For **Time-7**, the buggy method *parseInto()* was called in all three trials. However, EvoSuite did not generate any failing tests, indicating that while the method was called, the bug was not exposed. This is because to trigger the bug, the method has to be called with Feb 29 on a non-leap year, while all test suites generated did not include this prefix. Instead, EvoSuite generated nonsensical inputs such as `/2i({p{]:$bw`, as shown in Figure 21.

```java
public void test19() throws Throwable {
   try {
       DateTimeZone.forOffsetHoursMinutes(0, -75837680);
       fail("Expecting exception: IllegalArgumentException");
   } catch (IllegalArgumentException e) {
       // Minutes out of range: -75837680
       verifyException("org.joda.time.DateTimeZone", e);
   }
}
```

Figure 22: EvoSuite-generated test for Time-8. This test triggers an `IllegalArgumentException` by calling `forOffsetHoursMinutes()` with an out-of-range minute value.

```java
public void test04()  throws Throwable  {
 DateTimeZone dateTimeZone0 = DateTimeZone.forOffsetHoursMinutes(
   45, 45);
 String string0 = dateTimeZone0.toString();
 assertEquals("+45:45", string0);
}
```

Figure 23: EvoSuite-generated test for Time-9. The method returns a malformed zone ID for an invalid input `(45, 45)` instead of throwing an exception, exposing a functional bug.

For **Time-8**, although EvoSuite successfully executed the method `forOffsetHoursMinutes()` in all trials, it failed to generate inputs within the specific range required to trigger the bug. The bug manifests only when the minute value is between `-59` and `-1`, which are valid-looking inputs that previously led to exceptions due to missing handling. Instead, EvoSuite produced extreme values such as `(0, -75837680)`, which resulted in exceptions unrelated to the core bug. As a result, the fault condition was never exercised, and no test exposed the bug.

For **Time-9**, a functional bug, EvoSuite produced inputs such as `forOffsetHoursMinutes(45, 45)`, which bypassed input validation and returned a malformed zone ID like `"+45:45"`. As shown in Figure 23, the generated test included an output-based assertion that confirmed the incorrect behavior. This demonstrates EvoSuite's ability to detect behavioral faults even when the method does not crash, making Time-9 a clear success case for functional bug detection through output validation.

```java

public void test09()  throws Throwable  {
 LocalTime localTime0 = LocalTime.now();
 Partial partial0 = new Partial();
 // Undeclared exception!
 try {
   Weeks.weeksBetween((ReadablePartial) localTime0,
       (ReadablePartial) partial0);
   fail("Expecting exception: IllegalArgumentException");

 } catch(IllegalArgumentException e) {
    //
    // ReadablePartial objects must have the same set of fields
    //
    verifyException("org.joda.time.base.BaseSingleFieldPeriod", e);
 }
}
```

Figure 24: Test generated by Evo-suite for Time-10. The function `weeksBetween()` has been called with the first parameter `localTime0`, which is a `ReadablePartial`. But the date is not February 29th.

For **Time-10**, EvoSuite generated 29 tests in each trial. Among all three trials, the buggy branch was executed by calling `monthsBetween()` and `daysBetween()` with the first parameter type as `ReadablePartial`, an example is given in Figure 24. However, none of them triggered the bug since they did not try the specific date input, February 29th. The line coverage remained high, ranging from 93.0% to 94.4%, with condition coverage at 97.8%.

When we check the process of the evosuite, we observed that the bug-triggering input involving Feb 29 (`MonthDay(2, 29)`) was

not exercised by a single debug test case. Similar performance was observed in three different trials.

```
public void test36()  throws Throwable  {
    StringReader stringReader0 = new StringReader("\nZoneChar: ");
    BufferedReader bufferedReader0 = new BufferedReader(stringReader0);
    ZoneInfoCompiler zoneInfoCompiler0 = new ZoneInfoCompiler();
    zoneInfoCompiler0.parseDataFile(bufferedReader0);
}
```

Figure 25: Test generated by Evo-suite for Time-11. The `zoneInfoCompiler0` was declared as a new `ZoneInforCompiler`, which will execute the buggy branch. But it is not under the double thread.

For **Time-11**, EvoSuite generated between 38 and 46 tests across three trials. Among all three trials, all of them have triggered the buggy branch. The buggy-branch-covered tests include a declaration of `ZoneInfoCompiler`, an example can be found at Figure 25, which executed the buggy branch, since the buggy line will be executed once creating a `new ZoneInfoCompiler()`. However, it can only be triggered if it is under double thread. Failing test was not found in all trials since no double thread ever been called. Line coverage varied from 36.6% to 54.2%, and condition coverage ranged from 35.2% to 52.8%.

By checking the process, we can observe plenty of tests that execute the buggy branch, i.e., either directly initialize `ZoneInfoCompiler` or call some functions that indirectly initialize `ZoneInfoCompiler`. However, among all three trials, no input that can possibility trigger the bug was been found.

```
public void test095()  throws Throwable  {
    MockDate mockDate0 = new MockDate(0, 0, 0, (-1940), 0, 2);
    LocalDate localDate0 = LocalDate.fromDateFields(mockDate0);
    localDate0.minusMonths(0);
}
```

Figure 26: Test generated by Evo-suite for Time-12. The `.fromDateFields(mockDate0)` was called, which will execute the buggy branch with input before year zero. Thus, this is expected to trigger the bug.

For **Time-12**, EvoSuite generated between 101 and 128 tests for LocalDate and LocalDateTime classes. Similarly, among all three trials, the buggy branch was executed. The buggy branch under `.fromDateFields()` was executed in only one out of three; the buggy branch under `.fromCalendarFields()` was executed in all three trials. However, only one trial contains bug-triggering input, which executes `.fromDateFields()` with input before year zero, which we can find in Figure 26. Line coverage reached up to 91.9%, while condition coverage was as high as 90.3%.

2.2 Assertion Generation

To complement EvoSuite's prefix generation, we used EditAs2 to generate assertions based on the most promising prefixes for each bug.

Time-1: Constructor Safety Enforcement Not Triggered. For Time-1, EditAS2 retrieved the assertion `assertEquals(0, values.length)` and edited it to produce:

```
assertEquals(partial0.secondOfMinute(),
    partial0.clockhourOfDay());
```

However, this assertion failed to compile because the method `secondOfMinute()` does not exist in the `Partial` class. As a result, the assertion was never executable, and could not contribute to fault detection.

Even if it had compiled, the assertion would still be semantically incorrect. The bug in Time-1 is not a miscalculated return value, but the absence of a required runtime exception. When the `Partial` constructor is given a `DateTimeFieldType[]` array that is unordered or contains unsupported duplicates, it should throw an `IllegalArgumentException`. In the buggy version, it fails to do so.

This missing safety check is demonstrated in the developer-written test `testConstructorEx7_TypeArray_intArray()` (Figure 14), which asserts that such malformed input must be rejected. A correct assertion here would be:

```
assertThrows(IllegalArgumentException.class, () -> {
    new Partial(invalidTypes, values);
});
```

Although EditAS2 is capable of synthesizing `assertThrows`, it failed to generate one for this case. This could be due to a lack of semantically similar training examples or a poor match in the retrieval-edit pipeline. Consequently, while EvoSuite generated effective prefixes that triggered the bug, EditAS2 failed to produce a usable assertion.

This case highlights a limitation of assertion generation in safety-critical contexts. When the bug stems from a missing runtime check, rather than a wrong value, assertion inference must move beyond equality templates to capture exception expectations. In Time-1, bug detection relied entirely on prefix generation.

Time-2: Insertion Conflict Not Captured by Assertion. For Time-2, EditAS2 generated the following assertion:

```
Assert.assertEquals(partial0.toString(),
    partial0.getField(0).toString());
```

This line also failed to compile because `Partial` does not implement a `toString()` method, and the variable `Assert` was undeclared. As a result, the test suite could not even be compiled.

More fundamentally, even a working version of this assertion would not have expressed the bug. The underlying fault in Time-2 arises from adding conflicting time fields—such as `clockhourOfHalfday` and `hourOfHalfday`—to the same `Partial` object using `with(...)`. These two fields are logically overlapping and should not coexist. In the patched version, such combinations correctly throw an `IllegalArgumentException`, but the buggy version fails to reject them.

A correct assertion should check for the missing exception:

```
assertThrows(IllegalArgumentException.class, () -> {
    partial.with(clockhourOfHalfday, 10)
           .with(hourOfHalfday, 5);
});
```

Despite EvoSuite successfully generating such prefixes, EditAS2 failed to produce a semantically aligned assertion. This illustrates that for domain-specific constraints—like temporal field compatibility—assertion generation requires a deeper understanding of type semantics, which EditAS2 does not yet possess.

Time-3: Assertion Generation Skipped Due to Lack of Triggering Prefix. For Time-3, EditAS2 did not produce any usable assertions. This is not a failure of the assertion generation process per se, but rather a consequence of EvoSuite's inability to generate a triggering prefix.

As described in the previous section, although EvoSuite achieved high line and condition coverage for the buggy methods, none of the generated test cases actually reproduced the daylight saving time bug. The bug requires a precise sequence of method calls—for example, calling `addHours(1)` followed by `addYears(0)`—under a specific chronology and time zone setting. None of the generated prefixes met these conditions.

Since EditAS2 operates only on prefixes that trigger faults, and no such prefix was available for Time-3, it did not attempt to generate or edit any assertion for this case. As a result, Time-3 remains a case where both prefix and assertion generation must be domain-guided to succeed.

For Time-4, the prefix in figure19 is used to generate the assertion. The generated assertion `Assert.assertEquals(0, partial1.getValue());` could not compile, thus failed to trigger the bug. The prefix creates a `Partial` object `partial0` from the current `LocalTime` using a `ZonedChronology`, then adds the `minuteOfDay` field using the with() method. The generated assertion checks the resulting `partial1` by asserting that `partial1.getValue()` has a value of 0. However, `getValue()` requires an argument to indicate the index, and calling it without argument causes a compilation error.

For Time-6, the prefix in figure20 is used to generate the assertion. The generated assertion

```
assertEquals(long0, years0.get().getTime());
```

failed to compile. This prefix adds a large number of years to a timestamp before year 0, and the result is supposed to be a timestamp after year 1582, so the addition passes both year 0 and the cutover time. The assertion compares the result of the addition to `years0.get().getTime()`, which is an incorrect method call because `Years.get()` requires a `DurationFieldType` argument and does not return a time value. Thus, the test failed to compile. Semantically, the comparison is also wrong. `long0` is the result of addtion, the value of `years0` that it is attempting to get is 2057 which is defined in the prefix. There is no meaning to compare these two values.

For Time-9, EvoSuite generated a test that included the assertion `assertEquals("+45:45", zone.toString())`, which returns a malformed time zone string (e.g., `"+45:45"`). The correct assertion style here is `assertEquals(expectedValue, result)`. EvoSuite does not infer expected behavior; it merely captures and reasserts the output it observes. Therefore, the assertion cannot be considered meaningful in the Oracle sense. The correct assertion depends on the fixed behavior, which may involve either throwing an exception or returning a different zone string. On the other hand, EditAs2 failed to improve the test. It retrieved an unrelated placeholder assertion (`Assert.fail()`) and edited it into `assertEquals("095", string0)`, which does not match either the buggy or fixed output. As a result, EditAs2 did not contribute to exposing the bug.

For Time-12, EditAS2 generates the following:

```
assertThat ( localDate0 , is ( nullValue ( ) ) );
```

There are two main problems in this assertion. First, it uses `nullValue()`, is a Hamcrest matcher provided by the org.hamcrest.Matchers class. However, if the necessary static import is missing, Java doesn't know what `nullValue()` means, which is the same as what we will see if we try to test this. We will get the compilation error since it is not supported by defects4j. Second, this generated assertion is the wrong assertion. We tried to fixed the assertion to `assertNull(localDate0)`, which is a failing test for both buggy version and fixed version.

2.3 Success Analysis

Time-1. Time-1 clearly demonstrates EvoSuite and EditAS2's strength when dealing with constructor-level safety checks. The constructor of `Partial` immediately throws an `IllegalArgumentException` when duplicate or misordered `DateTimeFieldType` entries are provided. This deterministic failure made it easy for EvoSuite to generate a prefix that reliably triggers the buggy behavior in all three trials. The relatively small and predictable input domain (i.e., arrays of enum-like time fields) allowed EvoSuite to quickly evolve inputs that reach and trigger the exception.

EditAS2 also performed well here. The retrieved assertion shared high structural similarity with the correct behavior, and after neural editing, it produced an assertion that—while not semantically ideal—still led to exception exposure. This synergy between reachable code, clear failure, and partially relevant assertion enabled full bug discovery for Time-1.

Time-2. Despite EvoSuite successfully reaching the buggy method `with(...)` in all trials, Time-2 highlights the limits of current assertion generation techniques. The method requires nuanced reasoning about the semantic compatibility of time fields (e.g., era vs. year), which is not easily expressed in typical retrieved assertions. While prefixes reliably triggered exceptions during test generation, EditAS2's neural editing pipeline failed to synthesize an assertion that reflected this semantic conflict.

The root cause likely lies in the difficulty of retrieving a sufficiently similar prefix from the training corpus. Unlike simpler numerical comparisons, the bug in Time-2 relates to structural incompatibilities in internal state. These require assertions with high semantic precision, which EditAS2 could not infer from surface-level similarity alone. As a result, Time-2 received full prefix coverage but zero success in assertion-based fault detection.

Time-4. The bug Time-4 lies in a frequently used and shallow method `Partial.with(...)` which is both easy to reach and prone to misuse. In all 3 trials, EvoSuite successfully generated test cases

that executed the buggy branch where a new `DateTimeFieldType` is added to an existing `Partial`. This is largely because the method accepts field type inputs, and the logic involving insertion and validation of field types is deterministic and not deeply nested. Additionally, EvoSuite often generates combinations of time fields, for example `minuteOfHour` and `minuteOfDay`, increasing the likelihood of creating a semantic overlap that leads to the bug. Since the bug manifests as an invalid state during object construction, it is easier for a generated test to reach and expose it through structural coverage. Thus, the bug's accessibility and validation logic contributed to EvoSuite's success.

Time-6. EvoSuite successfully generated tests that execute the buggy branch in Time-6 across all 3 trials. This is due to the straightforward accessibility of the affected code. The bug resides in the `add(...)` method of `GJChronology`, which is a commonly invoked method when manipulating `DateTime`, `Years`, or other time-related fields across chronologies. The faulty condition involves improper handling of transitions across the Gregorian-Julian cutover, especially around year 0. Since EvoSuite readily generates boundary inputs, such as large or negative years and timestamps around the cutover date, it naturally exercises the conditional logic that reveals the bug. Therefore, the bug is successfully reached by Evosuite.

Time-9. Time-9 was successfully detected because several favorable conditions aligned. The method `forOffsetHoursMinutes(int, int)` accepts two integer inputs with relatively small and bounded domains, which made it easier for EvoSuite to explore edge cases like `(45, 45)`. The method did not throw exceptions but instead returned a malformed time zone string, which made the bug detectable through output-based assertions. EvoSuite generated a meaningful prefix and included an `assertEquals` assertion that matched the faulty output. Although EditAs2 produced an `assertEquals(...)` statement, the asserted value did not match the actual or expected output, and thus did not contribute to detecting the bug.

Time-12. For Time 12, the EvoSuite successfully generates a test with a bug-triggering input, which contains the value year before zero, and calling method `.fromDateFields()`. The bug occurs in a narrow part of the logic—specifically in how `LocalDate` and `LocalDateTime` parse date fields from legacy Date or Calendar objects. The methods involved are `.fromDateFields()` and `.fromCalendarFields()`, both of which are isolated, easily targetable entry points.

2.4 Error Analysis

Why did EvoSuite fail to trigger some bugs?

- **Bug not in test cluster**: If EvoSuite was unable to generate meaningful test inputs that reached the bug, this may indicate that the bug was located in a rarely executed path.
- **Search space complexity**: If the method had deep conditionals, complex logic, or external dependencies, EvoSuite may have struggled to generate the correct combinations of inputs to explore the relevant paths.
- **Randomness in test generation**: The variability across trials suggests that some paths were only covered under certain conditions. This highlights the impact of randomness in EvoSuite's test generation process.

Detailed Analysis. Although the buggy method `with(...)` in Time-2 was consistently executed in all three EvoSuite trials, the bug was not exposed by any generated assertion. The test inputs included invalid combinations of fields (e.g., `clockhourOfHalfday` and `hourOfHalfday`), which correctly triggered exceptions when run. However, the neural edit process failed to produce an assertion that checked for this failure. The final assertion instead compared unrelated string values, which neither triggered nor verified the presence of the exception.

This illustrates a core limitation of EditAS2 when applied to logic-heavy methods involving domain-specific constraints. Unlike more mechanical bugs (e.g., off-by-one errors or no-op additions), Time-2 requires an understanding of the time field hierarchy and rules around valid combinations—knowledge that is unlikely to be encoded purely through structural similarity. As a result, the combination of prefix and assertion was insufficient to find the bug, despite high coverage.

EvoSuite failed to generate a test that triggers the bug in Time-5 in all 3 trials. This failure is primarily due to the input-dependent nature of the bug and the lack of input diversity in EvoSuite's strategies. The bug in `normalizedStandard(PeriodType type)` only manifests when the provided `PeriodType` does not support either the year or month field, while the Period object itself contains values for those fields. However, EvoSuite consistently generated `PeriodType` inputs that included both year and month, avoiding the invalid configuration entirely. This highlights a major limitation in EvoSuite's exploration heuristics: it tends to favor safe, common constructor arguments and minimal field variation, rarely constructing custom or edge-case input combinations. This also shows a challenge for automated test generation: bugs that depend on field-level semantic mismatches are difficult to detect without domain-aware strategies.

The assertion generated by EditAs2 for Time-4 failed to trigger the bug primarily due to a compilation error, which prevented the test from executing at all. Specifically, the assertion Assert.assertEquals(0, partial1.getValue()); attempts to call the `getValue()` method without arguments, while the `Partial` class requires an integer index parameter to access the value of a specific field. This syntactic misuse of the API reveals a key weakness in EditAs2's assertion generation: it lacks type-aware validation, causing it to produce invalid method calls. Furthermore, even if the method had been called correctly, the assertion would still need to reference the correct index corresponding to the newly added `minuteOfDay` field in order to verify the bug-triggering behavior. Since the bug in Time-4 stems from incorrect field ordering due to a constructor argument swap `Partial(iChronology, types, values)` instead of `Partial(types, values, iChronology)`, the assertion would need to observe misaligned field semantics, not just raw values. Thus, both the syntactic error and the lack of semantic targeting contributed to the failure of EditAs2 to detect the bug. The assertion generated by EditAs2 for Time-4 failed to trigger the bug primarily due to a compilation error, which prevented the test from executing at all. Specifically, the assertion Assert.assertEquals(0, partial1.getValue()); attempts to call the getValue() method without

arguments, while the `Partial` class requires an integer index parameter to access the value of a specific field. This reveals the same weakness as the case of Time-4 did.

Time-7 was not detected because EvoSuite failed to generate a prefix that triggered the bug. The method `parseInto()` must be called with a leap day input such as `"2001-02-29"` on a non-leap year to exercise the faulty logic and produce an `IllegalFieldValueException`. Although `parseInto()` was executed in all three trials, EvoSuite only generated arbitrary, unstructured strings such as `/2i({p{]:$bw` and `,2El9"7T-&`, none of which resembled valid date formats. As confirmed by the debug logs, no inputs close to a leap day were used in any call to the method. Since the bug-triggering condition was never satisfied, the assertion generation phase was irrelevant. This case illustrates a failure in prefix generation due to the vast input space and lack of semantic guidance for generating calendar-sensitive strings.

Time-8 was not detected because EvoSuite failed to generate inputs within the narrow range required to trigger the bug. The method `forOffsetHoursMinutes()` is expected to throw an `IllegalArgumentException` for minute values between `-59` and `-1`, which were mishandled in the buggy version. However, EvoSuite only explored extreme values such as `(0, -75837680)`, which triggered unrelated exceptions but did not exercise the buggy behavior. As confirmed by the debug log, the closest observed value was `-75837680`, which is far outside the intended test space. No inputs near the valid-but-buggy range (e.g., `-15`) were generated. Since the faulty path was never reached, assertion generation could not contribute. This case highlights a weakness in prefix generation when subtle edge cases require semantically meaningful inputs.

For Time-10, EvoSuite did not successfully generate a test that could trigger the bug. The bug in Time-10 only manifests when `monthsBetween()` or `daysBetween()` is called with a `ReadablePartial` input representing February 29th. However, none of the generated tests used this specific date input, so the exception (`IllegalFieldValueException`) was never thrown, despite the buggy branch being executed. Because the bug relies on a very specific date input (`MonthDay(2, 29)`), EvoSuite failed to explore this narrow case within its search space. The input space for date values is large, and without specialized guidance or input constraints, it is unlikely to randomly select a rare edge case like February 29th. As a result, although line coverage remained high (93.0%–94.4%) and condition coverage reached 97.8%, EvoSuite was not able to trigger the functional property violation caused by initializing the chrono's start to 0L instead of starting from the leap year 1972.

For Time 11, EvoSuite did not successfully generate a test that could trigger the bug. The bug in Time 11 only manifests when code involving `ZoneInfoCompiler` is run in a second thread. However, none of the generated tests created or used a separate thread, so `ThreadLocal<Boolean>` always returned the correct default `(Boolean.FALSE)` from the static initializer of the main thread. Because it relies on a multi-threaded execution context, which EvoSuite failed to generate. The search space for concurrency-related bugs is inherently large and complex, and without explicit thread handling in the test cluster, EvoSuite could not trigger the faulty behavior—even though it executed the buggy code.

For Time 12, the EditAs2 did not successfully generate the right assertion, it did not cover the bug because:

(1) It has the wrong logic for checking if that is null or not. The right logic should be checking if the value for that is the same as expected or not, which would be `assertEquals(0, localDate0.getYear());`.
(2) Also, since there does not exist a valid `nullValue()` that defects4j supported, which is the reason that causes the compilation error.

The edit distance from retrieved distance is not large, changing with small edit distance from `assertThat(date, is(nullValue()))` to `assertThat(localDate0, is(nullValue()))`, which indeed converts the `date` to the right variable name `localDate0`.

3 IMPROVEMENT

3.1 EditAS2 - Domain-Aware Assertion Templates for Safety Properties

Our error analysis reveals a recurring shortcoming in EditAS2's assertion generation: it fails to synthesize meaningful assertions when the underlying bug stems from a missing runtime check (e.g., a missing `IllegalArgumentException`). This limitation was especially evident in **Time-1** and **Time-2**, where the faulty behavior was not a returned value mismatch, but a violation of a safety contract that should have been enforced by throwing an exception.

In both cases:

- EditAS2 retrieved a structurally similar assertion but edited it into a semantically invalid or syntactically incorrect form.
- The correct oracle required checking that a method throws an exception on invalid input (e.g., `assertThrows(IllegalArgumentException.class, ...)`), which EditAS2 failed to produce.

Meanwhile, in **Time-3**, EditAS2 did not generate any assertion at all—not because of a failure in the assertion engine itself, but because no triggering prefix was available due to EvoSuite's limitations. This highlights the importance of distinguishing between the responsibilities of prefix and assertion generation.

Proposed Solution. To address the core failure in **Time-1** and **Time-2**, we propose augmenting EditAS2 with a **domain-aware fallback mechanism** that applies exception-checking templates when:

(1) A prefix results in an unhandled runtime exception during test execution;
(2) The retrieved test case includes exception-handling logic (e.g., a try/catch block or failure on a known exception);
(3) The method being tested belongs to a known class with explicit safety contracts (e.g., `Partial`, `MutableDateTime`).

In such cases, EditAS2 should favor inserting the following pattern over standard `assertEquals(...)` statements:

```
assertThrows(ExpectedException.class, () -> {
    // prefix invocation
});
```

This strategy would ensure that safety violations—where the fault is the *absence of an exception*—are explicitly tested for.

Impact on Benchmarks. **Affected bugs:**

- **Time-1**: Assertion would change from an incorrect `assertEquals(...)` call to a correct `assertThrows(...)` assertion, thereby enabling EditAS2 to detect the bug using the already-valid prefix.
- **Time-2**: Instead of a syntactically invalid comparison between unrelated fields, the fallback mechanism would correctly assert that the conflicting insertion throws an exception.
- **Time-4**: Previously a wrong and meaningless assertEqual is called on two partials that should not be equal. With the modification, the assertion would be asserting on the validity of the created `Partial` instance.
- **Time-6**: The original assertion incorrectly assumed a specific timestamp output. However, the underlying issue was that the method silently computed an invalid Gregorian date across the Julian-Gregorian cutover. Our modification will replace the return-value assertion with `assertThrows(...)` to enforce the missing safety check.

Unchanged bugs:

- **Time-3**: No impact, as no prefix in any trial triggered the bug. Assertion generation remains inactive due to lack of input.
- **Time-5**: No impact, as this is a functional bug. The method executes without crashing but returns an incorrect value, which is outside the scope of our safety-oriented fallback.
- **Time-7**: No impact, as no bug-triggering prefix was generated. EditAs2 was not invoked to generate an assertion.
- **Time-8**: No impact, same as Time-7. The bug is a safety violation, but no prefix was available to expose it via EditAs2.
- **Time-9**: No impact, as this is a functional bug. The method completes execution but yields a semantically incorrect result.
- **Time-10**: No impact, also a functional bug. The method produces incorrect behavior, but no exception is expected.
- **Time-11**: No impact, as this is a functional bug with valid execution but incorrect output.
- **Time-12**: No impact, same as above. Our modification targets missing runtime checks, not return-value mismatches.

Justification. This improvement targets a specific failure mode observed in multiple bugs and addresses it with a lightweight, rule-based fallback. It requires no retraining of the EditAS2 model but can be integrated as a post-edit or pre-validation step. The proposal directly leverages our error analysis and provides a path to improving assertion effectiveness in safety-critical scenarios—precisely where accurate oracles are most important.

3.2 EvoSuite - Domain-Aware Input Seeding for Edge-Case Detection

Our error analysis of **Time-7** and **Time-8** and **Time-5** highlights a recurring limitation in EvoSuite's prefix generation: its failure to generate semantically meaningful inputs for structured domains like date strings or bounded numeric parameters. In both cases, the bug was not exposed due to missing relevant inputs, not a lack of assertion logic.

In **Time-7**, the method `parseInto()` requires inputs such as `"2001-02-29"` on a non-leap year to trigger an `IllegalFieldValueException`, but EvoSuite only produced arbitrary strings like `/2i({p{]:$bw` and `,2El9"7T-&`. In **Time-8**, the method `forOffsetHoursMinutes()` throws on valid-looking minute values between `-59` and `-1`, but EvoSuite generated only extreme values such as `-75837680` that did not trigger any failures.

Proposed Solution. To address these issues, we propose a lightweight extension to EvoSuite's prefix generation strategy: allow the developer to provide domain-aware seed inputs for specific methods. These seeds can be registered via a static configuration file or annotations and used to guide EvoSuite's initial input generation. For example:

- For date-parsing methods like `parseInto()`, EvoSuite should be seeded with semantically meaningful date strings, including edge cases such as leap days and year-end boundaries (e.g., `"2001-02-29"`, `"Feb 29"`, `"Dec 31 23:59"`), to increase the likelihood of triggering calendar-related parsing bugs.
- For integer-based methods like `forOffsetHoursMinutes()`, EvoSuite should prioritize small, signed values such as `-15`, `0`, and `59` for the minute parameter.

Similarly, in **Time5**, in order to generate semantically meaningful inputs with all kinds of PeriodTypes that can probably trigger the bug, if some seed inputs for different methods can be provided, EvoSuite's input generation will be stronger. Instead of always calling the method with a null instance of the required parameter type or just calling with empty constructor, it will be able to generate some more complex input.

This approach requires no modification to EvoSuite's core test generation logic. It simply adds an optional, externally configured input seeding phase that is compatible with EvoSuite's existing mutation and evolution pipeline.

Impact on Benchmarks. **Affected bugs:**

- **Time-7**: Seeding with valid leap-day strings would increase the chance of exercising the buggy input condition in `parseInto()`.
- **Time-8**: Prioritizing small negative minute values would increase the chance of reaching the faulty bounds-check logic in `forOffsetHoursMinutes()`.

Unchanged bugs:

- **Time-9**: EvoSuite already generated a bug-triggering prefix, so no additional input seeding would be necessary.

Justification. This improvement is lightweight, targeted, and easy to implement in future work. It avoids changes to EvoSuite's internal architecture by relying on developer-provided or preconfigured seed inputs. It directly addresses the failure mode observed in our error analysis for Time-7 and Time-8, and offers a generalizable mechanism for improving test coverage in domains where inputs must conform to specific formats or bounded ranges.

3.3 EvoSuite with Developer-Test-Guided Initialization

We propose enhancing EvoSuite's test generation by incorporating insights from existing developer-written tests when available. This approach differs from general edge-case detection by specifically leveraging known important values and patterns from human-written tests to guide the search process.

Motivation: Beyond Pure Randomness. Our analysis of Time-10 and Time-11 revealed limitations in purely stochastic generation:

- **Time-10 (Edge Case Dependency)**: While EvoSuite achieved 97.8% condition coverage on `monthsBetween()`, it missed the critical `MonthDay(2, 29)` input. Developer tests often explicitly include such temporal edge cases, suggesting they could inform the generation process.
- **Time-11 (Context Sensitivity)**: EvoSuite exercised `ZoneInfoCompiler` code but failed to simulate the required multi-threaded context—a design consideration more likely to appear in developer tests than emerge randomly.

These cases demonstrate that when developer tests exist, they contain valuable domain knowledge that could bootstrap EvoSuite's effectiveness.

Proposed Approach. Our solution operates in two phases when developer tests are available:

(1) **Test Analysis Phase**:
   - Extract constants (magic numbers, boundary values) and frequent object constructions
   - Identify common call sequences and exception handling patterns
   - Detect concurrency primitives (e.g., `ThreadLocal` usage)

(2) **Generation Guidance Phase**:
   - **Seed Initialization**: Use extracted values to bias the initial test population (e.g., prioritizing February dates for time APIs)
   - **Cluster Specialization**: Pre-initialize the test cluster with objects and method chains observed in developer tests
   - **Context Simulation**: For concurrency bugs, inject synchronization points or thread scheduling hints

Implementation Considerations.

- **Fallback Mechanism**: When no developer tests exist, EvoSuite defaults to standard random generation
- **Hybrid Mode**: Combine mined constants with evolutionary search (e.g., mutating developer-test-derived values)
- **Precision Anchoring**: For object-intensive APIs, use developer test sequences to "warm start" complex object constructions

This approach complements edge-case detection by providing known-important starting points for the search, particularly valuable for: (1) domain-specific constants (like February 29th), and (2) context-dependent bugs requiring precise initialization states. The key innovation lies in transforming developer tests into probabilistic guidance rather than hard constraints, preserving EvoSuite's exploratory nature while reducing wasted search effort.

4 IMPLEMENTATION

Our improvement targets the **assertion generation** phase of EditAS2. Specifically, we designed a fallback mechanism to detect and improve assertions for safety bugs where EditAS2 either produced incorrect assertions (e.g., invalid `assertEquals`) or failed to generate any meaningful oracle. Our strategy wraps bug-triggering method calls in `assertThrows(IllegalArgumentException.class, ...)` when a safety contract is violated but not enforced by the method under test.

4.1 Fallback Strategy

Our Python-based implementation consists of two main components:

- **Risky method detection:** We scan method source code using regular expressions to detect explicit `throw new ...` statements and `throws` clauses. For methods that throw exceptions implicitly through domain logic (e.g., `Partial.with`, `GJChronology.add`), we manually annotate them as risky. This manual step is sufficient for our proof-of-concept scope.
- **Test case rewriting:** Given an EvoSuite-generated prefix, we locate the bug-triggering line and wrap it in an `assertThrows(...)` block. Any invalid follow-up assertions, such as incorrect `assertEquals`, are removed.

This fallback is applied only to prefixes that successfully execute the buggy behavior. We do not modify the prefix generation process or re-run EvoSuite.

4.2 Evaluation Setup

We tested our fallback mechanism on five safety-related bugs where the prefix triggered the fault:

- **Time-1**, **Time-2**, **Time-4**, and **Time-10**: The assertion generated by EditAS2 was incorrect or meaningless.
- **Time-6**: The original test expected a specific timestamp, but the real issue was an invalid Gregorian date silently returned by `GJChronology.add`.

We reused the same prefixes and focused only on transforming the assertions. We did not re-run EvoSuite.

4.3 Colab Notebook

To support reproducibility, we provide a Google Colab notebook that contains our full implementation, test cases, and transformed results for each affected bug:
Colab link: You can find our colab link *here*

4.4 Discussion

4.4.1 Success Analysis. For **Time-1**, EditAS2 produced an invalid `assertEquals(...)` comparing unrelated fields on a `Partial` object. However, the actual bug stemmed from the constructor throwing an `IllegalArgumentException` due to duplicate field types. Our fallback successfully replaced the broken assertion with `assertThrows(IllegalArgumentException.class, ...)` wrapping the constructor, thereby producing a valid oracle that captures

Bug Number	Assertion (Original)	Assertion (Improved)	Manual Annotation?
1	Invalid `assertEquals` on unrelated fields	`assertThrows(IllegalArgumentException)`	No
2	Syntactically invalid assertion on `toString()`	`assertThrows(IllegalArgumentException)`	Yes
4	Irrelevant equality check between partials	`assertThrows(IllegalArgumentException)`	Yes
6	Misleading value assertion on invalid timestamp	`assertThrows(IllegalArgumentException)`	Yes
7	No assertion generated (prefix safe)	Inserted `assertThrows(...)` → false positive	No
8	No assertion generated (prefix safe)	Inserted `assertThrows(...)` → false positive	No
10	Missing assertion for date validation	`assertThrows(IllegalFieldValueException)`	No

Table 3: Assertion comparison between EditAS2 and our fallback mechanism. Manual annotation indicates cases where risky method identification was not inferred automatically.

the fault. This case was fully automated, as our tool detected the exception logic without requiring any manual annotation.

In **Time-2**, the test invoked `Partial.with(...)` with a conflicting field type. The original assertion generated by EditAS2 was syntactically malformed and semantically irrelevant. We manually annotated `with()` as a risky method and our fallback tool inserted a correct `assertThrows(...)` block around the method call, making the test meaningful.

Time-4 involved a subtle failure where calling `with()` internally constructed a `Partial` object with an invalid combination of field types. EditAS2 again generated an irrelevant `assertEquals(...)` on partial values. Our tool successfully identified the constructor invocation and replaced the faulty assertion with an exception oracle, correcting the test behavior.

Time-6 centered around `GJChronology.add(...)`, which silently returned an invalid Gregorian timestamp due to incorrect cutover handling. The test originally asserted a specific return value, masking the underlying safety violation. We manually marked `add()` as risky and applied our fallback, which replaced the value-based assertion with `assertThrows(IllegalArgumentException.class, ...)`. This exposed the missing runtime check and transformed the test into a valid bug oracle.

Finally, for **Time-10**, the original test created a BC date but failed to detect the leap year bug. Our tool automatically enhanced it by analyzing that `fromDateFields()` could throw `IllegalFieldValueException`. The improved test now properly validates the safety violation where February 29th dates should throw exceptions in non-leap years. This case demonstrates our tool's ability to handle temporal edge cases without manual intervention, transforming the test into an effective bug oracle that catches the calendar arithmetic fault. Unlike Time-1 through Time-6 which involved constructor validation, Time-10 shows successful detection of complex date boundary violations.

4.4.2 Error Analysis. In **Time-7**, the method `parseInto(...)` includes an explicit `IllegalArgumentException` throw when passed a null `instant`. Our fallback tool detected this and inserted an `assertThrows(...)` wrapper around the call. However, the EvoSuite-generated prefix passed a valid, non-null argument, so no exception was expected or thrown. This led to a false positive: the test failed not because of a bug, but because the fallback incorrectly predicted an exception. This case highlights a key limitation of our approach—without input-aware reasoning or symbolic analysis, the tool may wrap method calls that are syntactically risky but semantically safe in the given context.

In **Time-8**, the method `forOffsetHoursMinutes(...)` includes several explicit checks for invalid input, such as out-of-range hour and minute values. While the method contains multiple `IllegalArgumentException` throws, the EvoSuite prefix used valid input that did not trigger any of these conditions. However, our fallback tool inserted an `assertThrows(...)` wrapper anyway. This resulted in a false positive, since the test case completed without exception. Similar to Time-7, this highlights a key limitation of our approach: detecting that a method can throw does not imply that it will throw in the current test context. Our tool lacks input-aware reasoning and therefore may overwrap method calls even when they are safe.

For **Time-11**, our tool detected potential `ThreadLocal`-related failures but couldn't create the required multi-threaded context to expose the bug. While EvoSuite executed the relevant code, all tests ran in the main thread where `cVerbose` was properly initialized, resulting in false negatives. This reveals a key limitation: our approach can identify thread-sensitive risks but depends on the test generator to provide the necessary concurrency contexts, which EvoSuite failed to do. Unlike Time-7/8's false positives from over-wrapping, Time-11 shows the opposite problem - missing bugs due to insufficient test context generation.

For **Time-12**, the fallback mechanism incorrectly assumed the BC date conversion should throw an `IllegalArgumentException`, leading to a false positive test failure. The tool failed to recognize that the `MockDate` construction (-1940 year) represented a valid 1941 BC date input that should process normally through `fromDateFields()`. Instead of verifying the correct proleptic year calculation (where 1 BC should become year 0), the test wrongly expected an exception. This mirrors the over-wrapping issue seen in Time-7/8, where the tool's purely syntactic analysis of throw declarations led to inappropriate exception assertions for semantically valid inputs. The core bug - mishandling of BC year conversions - remained undetected because the test needed value assertions like `assertEquals(0, localDate0.getYear())` rather than exception checks. This case particularly highlights the challenge of distinguishing between legitimate exception cases and normal boundary value processing in date/time logic.

4.4.3 Limitations. A key limitation of our current implementation is that it always inserts `assertThrows(IllegalArgumentException.class, ...)`

regardless of the actual exception type. While this matched the cases we targeted in Defects4J, it overfits to the benchmark and would miss cases involving other common exceptions such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, or `ArithmeticException`. Furthermore, our prototype depends on manually annotating a small number of methods as "risky." Although this approach was sufficient for controlled evaluation, it limits generalization and reproducibility. Our implementation uses regular expressions for parsing Java code and test cases. While this is sufficient for prototyping, it is brittle and may fail on edge cases involving formatting or nested structures. A more robust implementation would use an AST-based parser to reliably analyze Java syntax and extract method-level behavior.

We also observed that our fallback mechanism can introduce false positives. Specifically, in Time-7 and Time-8, our tool inserted `assertThrows(...)` wrappers around method calls that did not actually trigger exceptions in the EvoSuite-generated tests. Although the methods under test do throw exceptions under invalid inputs, the specific test inputs used were valid, and the exception paths were not reached. This demonstrates that without input-aware reasoning, the fallback may overapproximate and incorrectly assert that exceptions should be thrown when they are not.

4.4.4 Generality and Future Work. Our approach is conceptually general and applicable beyond the specific bugs we studied. Many real-world systems rely on implicit safety contracts that are not enforced by visible exception handling. A fallback strategy that detects semantic violations and inserts appropriate exception-based assertions can strengthen automated test generation pipelines. To broaden applicability, future work should include automatic exception inference. This could involve extracting `throws` clauses from methods and their callees, weighting observed exceptions by frequency, or querying a language model for likely exception types. Implicit faults such as null dereferences could be estimated using static code features like field accesses. In addition, manual risk annotations could be replaced with heuristic detection or static analysis of exception-prone operations.

4.4.5 Cost-Benefit Tradeoff. Our fallback mechanism is lightweight and adds no runtime or memory overhead. It operates as a post-processing script on already-generated test code and does not require re-running EvoSuite or modifying the source under test. Despite its simplicity, the improvement turned several broken oracles into valid, bug-detecting tests. The correctness gains easily justified the minimal effort required to annotate a few risky methods. This suggests that small, targeted enhancements in assertion logic can meaningfully increase the utility of generated test suites.

5 CONCLUSION

We proposed a fallback mechanism to improve assertion generation in EditAS2 by wrapping safety-sensitive method calls in `assertThrows(...)`. Our tool automatically fixed one safety bug (Time-1) and successfully handled three more (Time-2, 4, 6) with manual annotation. However, it produced false positives on Time-7 and Time-8, highlighting the limitations of static exception detection without input-aware reasoning. Despite its simplicity, the fallback enhanced EditAS2's ability to expose safety bugs and offers a promising direction for improving test oracle quality in automated testing.

6 TEAM RESPONSIBILITIES

We divided the bugs as follows:

Time 1-3 - Tianyun Song
Time 4-6 - Yang Wu
Time 7-9 - Cecilia Zhang
Time 10-12 - Cecilia Chen